Chapter 3

The approXQL Query Language

This chapter provides a tutorial-style introduction to the approXQL query language. We distinguish between the *core syntax* and the *extended syntax* of the language. The design of the core syntax is based on the assumption that typical users have only partial knowledge of the structure of the XML documents, and are not willing (and often not able) to cope with a complex syntax. A query formulated using the core syntax consists of selection conditions on both the content and structure of the documents. These conditions can be combined with containment operators and the Boolean operators "`and`" and "`or`". The extended syntax provides primitives to modify the default semantics of the language and to specify typed selection conditions. It addresses users who have additional knowledge about the structure and content of the collection to be queried, and who are familiar with the non-standard semantics of approXQL.

We introduce the core syntax in Section 3.1 and the extended syntax in Section 3.2. The grammar of approXQL used in this chapter is simplified; we provide a full specification in Appendix A. To define the production rules of the grammar, we use the Extended Backus-Naur Form (EBNF) proposed by N. Wirth [Wir77]. In EBNF, [] brackets indicate zero or one occurrences of the enclosed expression; { } brackets indicate zero or more occurrences.

3.1 The Core Syntax

The core syntax is designed for users who have minimal knowledge about the structure of the collection to be queried. To formulate a query, the users must know the names of elements and attributes they are interested in, and the hierarchical relationships between these elements and attributes. This knowledge does not need to be exhaustive because the approXQL query processor can use value changes to vary element and attribute names, and permutations to vary the hierarchical relationships specified in the query.

The constituents of the core syntax are structural selectors, data selectors, predicates, containment expressions, and Boolean expressions. In the following, we successively introduce these constituents.

3.1.1 Structural Selectors

A structural selector such as `composer`, `title`, or `year` matches the name of an attribute or element. The users do not need to know whether a name appears as an attribute or element name in the XML documents. However, if the users know how a name is modeled, they can use a type prefix, as we describe in Section 3.2.3.

A structural selector is the simplest kind of query, as the partial syntax diagram indicates:

```
         Query  ::=  StructSelector
StructSelector  ::=  NAME
```

A query is executed against a collection of XML documents. It selects a set of *logical documents*. A logical document may be an element or an attribute of an element[1]. The concept of logical documents allows us to cleanly differentiate between physical XML documents, which are defined by the document creator, and logical documents, which are defined at query time. We say that the name of an element or attribute is the *root value* of the logical document defined by this element or attribute. Consider Figure 3.1 on the facing page. There are two physical XML documents but 18 logical documents, because there are two attributes and 16 elements altogether.

We use the term *root selector* to denote the topmost selector of a query. The root selector determines the candidate set of logical documents to be retrieved as results. Thus, the query

```
cd
```

selects all logical documents with the root value `cd`. In the collection depicted in Figure 3.1 two results of this query exist.

[1] We give an alternative definition of logical documents in Chapter 4, where a collection of XML documents is interpreted as a data tree, and a logical document is a subtree of the data tree

```
<catalog>
  ...
  <cd year="2001">
    <performer>Ashkenazy</performer>
    <composer>Rachmaninov</composer>
    <title>Piano concerto no. 1</title>
    <tracks>
      <track length="13:25">
        <title>Vivace</title>
      </track>
      ...
    </tracks>
  </cd>
  ...
</catalog>
```

```
<catalog>
  ...
  <cd year="1998">
    <category>Classics</category>
    <performer>Rachmaninov</performer>
    <title>A window in time</title>
    <tracks>
      <track length="5:42">
        <title>Piano sonata</title>
      </track>
      ...
    </tracks>
  </cd>
  ...
</catalog>
```

Figure 3.1: Two physical XML documents, each including a logical document with the root value `cd`.

3.1.2 Data Selectors and Predicates

Data selectors match *tokens* within text sequences and attribute values of XML documents. A token is a substring of a text sequence or attribute value that is separated from other tokens by whitespaces. In the core syntax, we distinguish between text tokens and numerical tokens. All text tokens in a query and in XML documents are stemmed and converted to lower case. A *text selector* is a special kind of data selector, which consists of text tokens enclosed by quotation marks. The strings

`"piano"` and `"piano forte"`

are valid text selectors. We call the right selector a *phrase selector*. It matches the specified words if they appear in a document in the given order, and without other words in between. A data selector preceded by an operator forms a *predicate*:

```
    Predicate  ::=  [ Operator ] DataSelector
 DataSelector  ::=  Phrase | NUMBER
       Phrase  ::=  '"' AlphaNum { AlphaNum } '"'
     AlphaNum  ::=  WORD | NUMBER
```

A `WORD` is a character sequence that starts with a letter and continues with letters and digits; a `NUMBER` is a sequence of digits, optionally separated by a decimal point. Operators for numerical data are '=', '<', '<=', '>', and '>='. For text data, only the test for equality is defined. For example, the sequence of literals

```
> 2000
```

is a predicate that matches numerical tokens whose values are greater than 2000. The production rule for predicates shows that the operator can be omitted. In this case, the default operator "`=`" is used. A lone predicate is *not* a valid query; however, it is one constituent of a query *expression*:

```
Expression  ::=  Query | Predicate
```

3.1.3 Containment Expressions

Queries may be composed to specify containment relationships, which must be fulfilled by the matching elements and attributes in the XML documents. A containment relationship is specified by brackets. For example, the query

```
cd[composer["rachmaninov"]]
```

selects CDs that contain works composed by Rachmaninov. To yield an (exact) match, the `cd` element must have an attribute or a directly contained element named `composer`. Additionally, the attribute value or the text directly contained in the `composer` element must include one or more text tokens with the value `rachmaninov`.

We do not demand that the inner parts of the query select numbers or words. The query

```
cd[review]
```

matches all CDs that have a review. We extend the formal definition of a query:

```
Query  ::=  StructSelector [ '[' Expression ']' ]
```

We call any query contained in another one a *subquery*. Each pair of brackets defines a *hierarchy level* of the query.

3.1.4 Boolean Expressions

So far, each hierarchy level consists only of a single subquery or a single predicate. To specify that a CD must have a year and/or a composer, approXQL provides the Boolean operators "`and`" and "`or`". For example, the query

```
cd[year[> 2000] and composer["rachmaninov" or "prokofiev"]]
```

requests CDs that appeared after 2000 and contain works composed by Rachmaninov or Prokofiev. Boolean expressions at the same hierarchy level may be arbitrarily nested as the following example shows:

```
cd[year[> 2000] and (composer["rachmaninov"] or performer["ashkenazy"])].
```

To give the full (although simplified) specification of the core syntax, we extend the definition of an expression, and add definitions for disjunctions and conjunctions:

```
         Query  ::=  StructSelector [ '[' Expression ']' ]
    Expression  ::=  Query | Predicate | Disjunction | '(' Expression ')'
   Disjunction  ::=  Conjunction { 'or' Conjunction }
   Conjunction  ::=  Expression { 'and' Expression }
StructSelector  ::=  NAME
     Predicate  ::=  [ Operator ] DataSelector
  DataSelector  ::=  Phrase | NUMBER
        Phrase  ::=  '"' AlphaNum { AlphaNum } '"'
      AlphaNum  ::=  WORD | NUMBER
```

To further simplify the syntax, we define an abbreviated notation for containment expressions: Whenever an expression enclosed by brackets consists of a single subquery or a single predicate, then a slash may be used instead of brackets. A containment expression written in simplified syntax resembles a path expression defined by the XPath standard [CD99]. The previous example query can be written as:

```
cd[year/>2000 and (composer/"rachmaninov" or performer/"ashkenazy")].
```

3.2 The Extended Syntax

The extended syntax addresses users who want to express their information needs even more precisely than it is possible with the core syntax. To use the extended syntax, the users must have good knowledge of the structure of the documents to be queried, and must be familiar with the non-standard semantics of approXQL. We introduce the formal model of the semantics in Chapter 5. However, to understand the effect of the extended syntax on the interpretation of a query, we sketch the main principles of the semantics.

For each logical document in a collection, the query engine tries to find an exact embedding of the query. An exact embedding is a mapping of query selectors to elements, attributes, and words such that the mapping preserves the values of the selectors as well as the containment relationships of the selectors. Consider the query

```
cd[title["piano" and "concerto"] and composer/"rachmaninov"].
```

It has an exact embedding in the left document, but not in the right document depicted in Figure 3.1 on page 33. The part of the left logical document matched by the query is drawn in black; all other document parts are grey.

Clearly, there may be logical documents that are semantically close to the query, but it is not possible to exactly embed the query. For each of those documents, the query engine tries to transform the query in a way that the resulting query has an exact embedding in the document. We allow the deletion, permutation, and insertion of query selectors, as well as the change of the values assigned to the selectors. Each basic transformation has a cost; the total cost of the transformation sequence determines the score of the document. Consider again our example query. To yield an exact embedding into the right document shown in Figure 3.1, the leaf selector `"concerto"` must be deleted, the selectors `tracks` and `track` must be inserted, and the value of the selector `composer` must be changed to `performer`. The resulting query

```
cd[tracks/track/title/"piano" and performer/"rachmaninov"]
```

can now be embedded exactly into the document. The black parts of the right document show the elements and text sequences matched by the transformed query.

Sometimes, the users may not want to allow all possible transformations. For instance, they may only want to find CDs that contain all specified keywords (and not a subset of them), and they may want the keywords to appear in the CD title (and not in a track title). They may also want to find sound storage media that are in fact composed (and not performed) by Rachmaninov. In other situations, the users may want to assign the same scores to results that have different structures. For instance, they may want to express that a query should treat CD titles and track titles equally: Both variants should get the same score, and thus the same position in the ranking.

We support both types of user preferences in our model. Any *restriction* requested by a user forbids the corresponding query transformation; any specified *relaxation* removes the cost from the corresponding transformation.

3.2.1 Query Restrictions

Restrictions forbid query transformations, or impose additional limitations on the query mappings. We support three types of restrictions that concern the insertion and deletion of selectors, as well as the change of the value of a selector.

An *insertion restriction* forbids the insertion of selectors into a certain part of a query. The users can express this kind of restriction by adding an exclamation mark as prefix to a selector whose matches are required to be directly contained in the elements matched by its parent selector. The query

```
cd[!title["piano" and "concerto"] and composer/"rachmaninov"]
```

shows an insertion restriction: Inserting selectors between `cd` and `title` is not allowed, which means that `cd` elements are only selected if they have `title` attributes or directly contained `title` elements. Using this syntax, we can simulate the XPath expression "`cd/title`".

A *value-change restriction* forbids the change of the value of a query selector. This kind of restriction includes both element and attribute selectors like `title`, and data selectors like `"rachmaninov"`. A value-change restriction is indicated by an exclamation mark that follows the concerned selector. For instance, the exclamation marks in the query

```
cd![title["piano" and "concerto"!] and composer/"rachmaninov"]
```

suppress value changes of the selectors `cd` and `"concerto"`.

A *deletion restriction* forbids the deletion of a selector. As with the other types of restrictions, a deletion restriction is denoted by an exclamation mark. The mark follows the selector it concerns, and is separated by a colon. The query

```
cd[title["piano":! and "concerto":!] and composer:!/"rachmaninov"]
```

shows an example, where the selectors `"piano"`, `"concerto"`, and `composer` must be retained. The three types of restrictions can be used in combination:

```
cd![!title["piano":! and "concerto"!:!] and composer:!/"rachmaninov"].
```

If all possible insertions, value changes, and deletions are explicitly forbidden, and if no permutations are defined, then the semantics of a "fully restricted" query is equivalent to the semantics of the same query phrased in a traditional XML query language, where containment relationships of the query are mapped to direct-containment relationships in the documents. If no restrictions are applied, the approXQL query processor still selects the exact results — but only as best matches among many other results ranked by decreasing similarity.

3.2.2 Query Relaxations

A relaxation removes the cost from a query transformation. We distinguish between *insertion relaxations*, which allow the inserting of any number of selectors without costs, and *value-change relaxations*, which allow the cost-free changing of the values of selectors.

An insertion relaxation is indicated by an asterisk in front of the selector whose matches may appear anywhere below the matches of the parent selector. The query

```
cd[*title["piano" and "concerto"] and *composer/*"rachmaninov"]
```

relaxes the insertion of selectors between `cd` and `title`, `cd` and `composer`, and `composer` and `rachmaninov`, respectively. Now, CD titles and CD track titles are cost-equivalent matches for the `title` selector. Similarly, the matches for the `composer` selector may appear at any distance from the matches of the `cd` selector, and the keyword `"rachmaninov"` may appear at any depth in a `composer` part of the document. The relaxation operator "`*`" of approXQL is exactly equivalent to the XQL operator "//": It skips any number of elements without imposing costs.

Value-change relaxations allow the cost-free mapping of a query selector to different values in the documents. All alternative values are assigned to the respective selector, separated by "`|`" signs. We uniformly allow the value change of the root selector, of inner selectors, and of leaf selectors. The query

```
(cd | mc | dvd)[title["piano" and ("concerto" | "sonata")] and
                (composer | performer)/"rachmaninov"].
```

shows all three cases. The query specifies that CDs, MCs, and DVDs have to be treated equally. Similarly, the query selects both piano concertos and piano sonatas, and retrieves works composed or performed by Rachmaninov without imposing costs. Note that the sign "`|`" applied to leaf selectors is semantically equivalent to an "`or`" operator.

3.2.3 The Type System

A structural selector in an approXQL query matches element and attribute names uniformly. Similarly, a numerical selector without a decimal point matches both integers and real numbers. To restrict the scope of a selector, approXQL supports a simple type system. The type system, together with query restrictions and relaxations, provides a means to adapt the query semantics to the user's knowledge of the application domain and expertise in query formulation. In this subsection, we informally introduce all syntax-related aspects of the type system. In Section 4.2, we provide a formal model of type-value trees, which represent queries and documents in a uniform way.

The XML standard does not define an explicit type system; it only defines a logical document structure that consists of elements, attributes, parsed text, non-parsed text, and further components.[2] The logical document structure is the basis for the type system of approXQL. There are two type hierarchies: The hierarchy of structural types consists of the general type *struct* and its specializations *element* and *attribute*. Figure 3.2 on the following page shows this type hierarchy. In contrast to elements and attributes, which are explicitly marked up, XML knows only parsed text and non-parsed text to represent the content of documents. To be able to detect data types automatically, we must rely on syntactic properties of text sequences. The standard data types supported by approXQL are *text*, *integer*, and *real*. Figure 3.3 on the next page shows the data type hierarchy. The standard types can be enriched by more specialized types like person name, date, timestamp, and so on. A precondition for the use of such extensions is that the types of the tokens in a document can be determined either automatically or with little human assistance. For name identification, several tools are available, e.g., those described in [Hay94, RW97]. The grey parts of Figure 3.3 show possible extensions of the standard data type hierarchy.

The use of the type system in approXQL queries follows our general philosophy: The default type of a selector is the most general type of the structural type hierarchy or the data type hierarchy, respectively. Our example query

```
cd[year[> 2000] and composer["rachmaninov"]]
```

[2]There are several proposals to enrich XML documents with data types, e.g., [DFH$^+$99, BCML99, BGP00]. Recently, the World Wide Web Consortium (W3C) recommended XML Schema [TBMM01, BM01] as the new standard for defining the structure, content and semantics of XML documents, including a rich set of data types. However, it cannot be expected that every XML document will be an instance of a schema; this holds in particular for semistructured documents. To be as general as possible, approXQL ignores every schema that is referenced in a document.

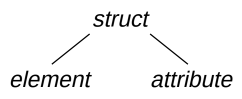

Figure 3.2: The structural type hierarchy.

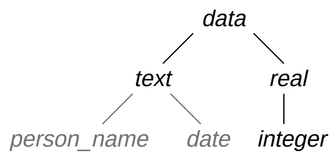

Figure 3.3: The default data type hierarchy and two extensions.

is an abbreviation for

```
struct:cd[struct:year[real > 2000] and
          struct:composer[text = "rachmaninov"]].
```

A selector or predicate of a general type matches all document parts of that type and all document parts with a more specific type. For example, the selector `struct:year` matches both elements and attributes with the value `cd`; the predicate `year > 2000` matches both integers and real numbers. By providing an explicit type, the users can restrict the scope of the selectors. For example, the selectors in the query

```
element:cd[attribute:year[integer > 2000] and
           element:composer[person_name = "rachmaninov"]]
```

match only `cd` elements, `year` attributes, `composer` elements, integers greater than 2000, and persons with the name Rachmaninov.